Homework #8

Released: 02-28-2017
Due: 03-07-2017 11:59pm

In the last assignment, we will practice class inheritance and subtype polymorphism. Instead of having a monolithic `Node` class representing all types of machines, we will split it into three subclasses, `Laptop`, `Server` and `WAN_node` to model the machines in finer granularity.

Please do not modify `datagram.h`, `machines.h` or the lines "`friend class Grader;`" in homework 8. These are needed for grading.

- The `Node` class now only serves as an interface and contains only two common data members `name_` and `local_ip_` and the implementation of one common member function `Node::get_ip`.

  A new member function has been added to all four classes, `Node::can_connect`, which checks whether two machines are able to connect to each other.

- The two member functions `allocate_datagram` and `release_datagram` now belong only to `Laptop`. The `Laptop` class is only able to connect to one server, thus we replace `node_list_` by a shared pointer to the `Server` class.

  The `Laptop` class doesn't have unbounded send buffer `data_list_` anymore. It contains only a single send buffer slot, `Datagram* outgoing_;`. Of course, a `Laptop` class needs to delete the datagram in `outgoing_` manually when it is destructed.

- The `Server` class and the `WAN_node` class are similar. The only difference is that a `Server` can only connect to `Laptop`s and `WAN_node`s while a `WAN_node` can only connect to `Server`s and other `WAN_node`s.

  Both `Server` and `WAN_node` are not allowed to be the destination of a `Datagram`, so they don't have the `incoming_` field anymore.

  A few data members such as `size_t num_servers_` and `size_t num_laptops_` have been added to these two classes to count the number of established connections.

1 Machine Types

Implement the member function `can_connect` for subclasses `Laptop`, `Server` and `WAN_node`. The member function `bool can_connect(const std::shared_ptr<Node>&) const;` checks whether the current machine can make a new connection to `x`.

Here, we impose the restrictions that a laptop can connect to exactly one server. A server can connect to at most 8 laptops and 4 WAN nodes. A WAN node can connect to at most 4 servers and 4 WAN nodes.

- Modify `System::connect_machine` to check the validity of a connection before actually connecting two machines. A connection between machines `m1` and `m2` can be created only when both `m1->can_connect(m2)` and `m2->can_connect(m1)` are `true`.

- Modify `System::create_machine` to create either the subclass `Laptop`, `Server` or `WAN_node` depending on the `type` argument, instead of the base class `Node`.

2 Starting and Ending Data Transmission

Modify `System::allocate_datagram` and `System::release_datagram` to down-cast the designated `Node` to `Laptop` and invoke `Laptop::allocate_datagram` and `Laptop::release_datagram`, respectively.

3 Sending Data

Split `send` and `receive` into the subclasses appropriately.

In the `Laptop` class, since it can only connect to exactly one server, it should send the datagram pointed by `outgoing_` to this server. The pointer `outgoing_` has to be set to `nullptr` after sending out the datagram it points to.

In the `Server` class, the behavior is roughly the same as the `Node::send` function in homework 7. Given a datagram, if the destination of the datagram is in `node_list_`, and if that destination is a `Laptop`, send the datagram to that machine. Otherwise, find the *`WAN_node` machine* in `node_list_` whose IP address's first octad is closest to the destination's first octad, and send the datagram to that `WAN_node`. Similar to homework 7, keep the datagram in `data_list_` if an `err_code::recv_blocked` exception is raised.

In the `WAN_node` class, the process is also close to `Node::send` in homework 7. However, since a `WAN_node` cannot connect to a `Laptop`, the destination of the datagram will not appear in `node_list_`. So we only find the machine whose IP address's first octad is closest to the destination's first octad. Also, since a `WAN_node` will never connect to a `Laptop`, an `err_code::recv_blocked` exception will never be raised when we call `Node::receive`.

4 Receiving Data

In homework 7, when a `Node` receives a datagram, it will check whether the destination of the datagram is the current machine. If it is, the machine will try to place the datagram in the receive buffer `incoming_`. If the destination is not the current machine, the datagram will be pushed to the back of `data_list_` instead.

In this homework, the algorithm becomes even simpler since the subclasses model the machines better. In `Laptop`, when the destination is the current machine, the incoming datagram is placed in the `incomine_` buffer. In `Server` and `WAN_node`, the current machine will never be the destination of the datagram. Thus, simply pushed to the back of `data_list_` in `Server` and `WAN_node`.

5 Handling Errors

For error handling,

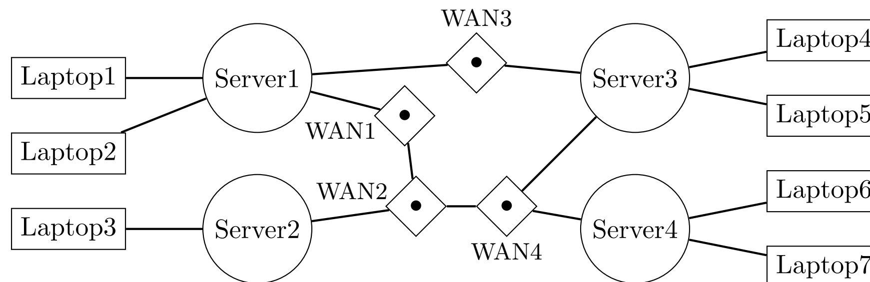
- In `Laptop::allocate_datagram`, if the send buffer `outgoing_` is not `nullptr`, raise an `err_code::send_blocked` exception.
- In `Laptop::connect`, if the designated machine is not an instance of the `Server` class, or if `server_` is not `nullptr`, raise an `err_code::connect_failed` exception.
- In `Laptop::receive`, in addition to throwing `err_code::recv_blocked` when `incoming_` is not `nullptr`, the `Laptop` should also raise an `err_code::recv_blocked` exception when the destination of the datagram does not match the IP address of the current machine.
- In `System::allocate_datagram`, the `System` should raise an `err_code::no_such_machine` exception if either the source machine or the destination machine is not a `Laptop`.
- In `System::release_datagram`, the `System` should raise an `err_code::no_such_machine` exception if the designated machine is not a `Laptop`.
- In `System::connect_machine`, the `System` should raise an `err_code::connect_failed` exception if either machine returns false in `Node::can_connect`.

6 Unit Testing

For unit testing,

- Adapt the unit tests about `Node` in homework 6 and homework 7 to use the three subclasses `Laptop`, `Server` and `WAN_node`.
- Add a new test to check that during `Server::send`, the datagrams whose destinations are not in `Server::node_list_`, will only be sent to other `WAN_node`s.
- Add new tests to check that in `System::connect_machine`, a connection can only be made if the types of the two machines are correct. Also check that an exception is thrown for incorrect machine types.
- Add a new test to check that in `System::allocate_datagram`, an exception is thrown when the destination machine is not a `Laptop`.

Appendix: Project Introduction: Homework 5-8



In this project, we are going to build a tiny network simulator modeling a small system that has laptops, servers and WAN (Wide Area Network) nodes. We will also model datagram transmission between them. A laptop must first be connected to a server. A server can connect multiple laptops, building a LAN (Local Area Network) between them. A server can also be connected to multiple WANs, in which case it will be able to transfer datagrams indirectly to other servers and finally to other laptops outside LAN. A WAN node can connect not only to arbitrary servers, but also to other WAN nodes.

Starting from homework 6, we will implement one class for each of the constructs in this system: a `System` class for the entire network system, a `Datagram` class for datagrams and machine classes `Laptop`, `Server`, `WAN_node` for laptops, servers, and WAN nodes respectively. The `System` class will have member functions corresponding to network operations. These include: sending and receiving a datagram on a `Laptop`, adding and removing machines from the network, and a time ticking function for servers and WAN nodes to route datagrams one step toward their destination.

The simulator, aside from the `System` class modeling the entire network, also contains a command line interface to interact with the user. The user can enter commands to control the system and view the status of the network system. In this homework 5, we implemented three utility parsing functions that help the command line interface convert input strings into commands and accompanying data in order to invoke the corresponding member functions of the `System` class.

In provided the code, `main.cpp` and `interface.cpp` implement the command line interface. In `main.cpp`, the `main` function repeatedly reads a line from the user, parses the input into tokens by the `tokenize` function, and calls `execute_command` to perform the corresponding operations. If an error is thrown, it catches the error code `err_code` and prints an error message.

In `interface.cpp`, the `execute_command` function first identifies the input command by searching through the `command_syntaxes` list, match the command string and obtain the `cmd_code` for the input command. `execute_command` then parses the accompanying data (some by `parse_IP` and invokes the member function of `System`.